3. Designing Enabling Technologies: The MOOSE Language and the MacMOOSE Client

3.1 The Need for a New Language

In March of 1994, I had the opportunity to visit the MariMUSE project in Phoenix, Arizona. The previous summer, Billie Hughes and Jim Walters of Phoenix College had led a summer program where children from Longview Elementary School used a MUSE intensively, three hours a day for two three-week sessions. Seven months later, the children were still excited about their projects, and regretted not having as much time to use the MUSE as they had in the summer. They had done some impressive creative writing and building of rooms. One fifth-grade girl built a palatial mansion with flowers in each room. In real life, she lives in a homeless shelter. A Native-American boy whom teachers had considered at risk of dropping out of school built an airplane hangar. To get more realistic detail, he had read every book about planes in his school library, and asked the librarian to order more (Walters and Hughes 1994).

The children had clearly had powerful learning experiences. But something was missing. The planes didn't fly. The flowers couldn't be sniffed. None of the objects had any behaviors. None of the children I met had done any programming. One girl had made a horse with a "ride" program, but it didn't work. I sat down to look at it with her–why don't we try to fix it, I asked? She replied that an adult had written the program for her, and she had no idea how it was supposed to work.

It's not a surprise that the children were having difficulties. The MUSE language is awkward at best. Suppose that you have created an object called Rover and you would like it to wag its tail when you pet it. In MUSE, you would write:

```
@va Rover=$PET ROVER:@pemit You pet Rover.; @emit Rover wags his
   tail.
```

The name of the object can not be abstracted, but is treated as a fixed string. If you would like to be able to type "pet dog" in addition to "pet Rover," you would need to add a second command:

```
@vb Rover=$PET DOG:@pemit You pet Rover.; @emit Rover wags his tail.
```

In MUSE, each object has twenty-six lettered registers, va through vz, and a set of special-purpose, functional registers. Not only must all programs be stored on these registers, but also all data.

Now suppose you decide you want Rover to be female instead of male. To change the "his" to "her" you could retype the two lines above, or use the @edit command:

```
@edit Rover/va=his,her
@edit Rover/vb=his,her
```

The syntax is obscure, and these substitutions are prone to error. In a more complex text, it's easy to pick too short of a target string to substitute and end up changing the wrong word (for example, changing "this" to "ther" when you meant to change "his" to "her" later on in the line.) The interface to working with the language is as much a barrier as the language itself.

Some of the children participating in MicroMUSE[1] and MariMUSE have indeed written programs. Here's a sample program written by an eleven-year-old girl on MariMUSE using the character name "Ginji" (Walters and Hughes 1994):

```
tardis You see a unusual looking stalagmite. Type 'enter tardis to be
   able to use it! Owner: Ginji Credits: 1 Type: Thing Flags: haven
   enter_ok visible quiet
Attribute definitions: scan return #13711.return:@fo #13711=@tel
   #13035;@remit here=You see the Tardis disappear!
Va:$tower:@fo #13711={@tel #1722;@wait 15=@tr me/return}
Idesc:You are inside of the tardis.
Pennies:1 #13711.scan:$-scan:@pemit [v(n)]=> Scanning Location
   [loc(me)];@pemit [v(n)]=>REPORT:;@pemit [v(n)]=[string(-
   ,77)];look;@pemit [v(n)]=[string(-,77)]
Listen:*
Vb:$stable:@fo #13711={@tel #12375;@wait 15=@tr me/return}
Vc:$recycle:@fo #13711={@tel#13073;wait 15=@tr me/return}
Lock:Ginji(#11059Pvn)
Contents: Avalon(#12Pevc)
Home:Gingi's Mystery Cave(#13035RHvJ)
Location: Gingi's Mystery Cave(#13035RHvJ)
```

Given this cryptic and obscure syntax, it's not a surprise that very few of the participants wrote programs. Good software can serve as a scaffolding to support children's learning experiences (Pea 1993; Guzdial 1994). Poorly-designed software acts more like a road block.

In early 1995, Phoenix College and Xerox PARC jointly received a grant from the Department of Defense Advanced Research Projects Agency (DARPA). The MariMUSE project was renamed Pueblo, and the software was changed from MUSE to MOO (Curtis 1993). MOO is significantly more user-friendly than MUSE, and by far the best of the many currently available MUD languages—users who are not trained programmers have been able to

---

[1]MicroMUSE is the oldest and largest MUD for kids. It was founded by Stan Lim in 1990, and leadership of the project was soon taken over by Barry Kort (Brown 1992). Its official charter states that "MicroMUSE is chartered as an educational Multi-User Simulation Environment (MUSE) and Virtual Community with preference toward educational content of a scientific and cultural nature" (ftp://ftp.musenet.org/micromuse/Mission.Statement).

achieve more in MOO than in any other MUD language. A script to pet Rover in MOO looks like this:

```
@verb Rover:pet this none none
@program Rover:pet
player:tell("You pet Rover.");
this.location:announce_all("Rover wags his tail.");
.
```

In MUSE, to be able to "pet dog" as well as "pet Rover," we needed to duplicate the program. In MOO, we can simply add "dog" as an alias for Rover. The word "this" in the verb declaration is an abstraction that the parser will match to any valid name for the object.

MOO, unlike MUSE, is a full programming language in which it's possible to undertake large, complex projects. Its object-oriented nature means children can quickly create something satisfying by making a new object that inherits from an existing parent object, and then customize it and add new functionality. You can create a dog by making something that inherits from generic dog, and then program it to do new tricks. Billie Hughes highlights this as the main benefit she has observed in moving from MUSE to MOO:

> Our children do not do lots and lots of programming though many do create simple verbs. We actually found teaching verbs to be much easier than we anticipated. One 6th grader took a "generic pet" and modified the code to create a generic horse. She then set this fertile and others could adopt horses.
>
> What we have found particularly exciting about Pueblo is the inheritance/parenting/child features of an object oriented language. Hobbes [Kim Bobrow, a researcher working on the project] was able to program cats, dogs, and cars that were especially popular. Kids also loved food and clothing items. MOO let them easily create these type of objects and change the messages on the object to personalize them (Hughes 1996).

While MOO is a significant improvement over MUSE and other MUD languages, it is still too complicated for most children. Its syntax resembles a cross between C and Pascal, and any deviation from that rigidly prescribed syntax triggers an often cryptic error message. Difficult concepts are required to write even the simplest programs. Something more user-friendly is needed if children are to master it.

The MOOSE language was designed to be forgiving. For example, it's possible to forget the quotes in programs. This only causes problems if the unquoted strings contain words used as operators in MOOSE (like "and" and "or"). In

that case, MOOSE is able to detect the problem and warn the user. Table 3.1 presents the same program in MUSE, MOO, and MOOSE.

MUSE
```
@va Rover=$PET ROVER:@pemit You pet Rover.; @emit Rover wags his
   tail.
@vb Rover=$PET DOG:@pemit You pet Rover.; @emit Rover wags his tail.
```

MOO
```
@verb Rover:pet this none none
@program Rover:pet
player:tell("You pet Rover.");
this.location:announce_all("Rover wags his tail.");
.
```

MOOSE
```
on[2] pet this
  tell player "You pet Rover."
  emote "wags his tail."
end
```

Table 3.1: Petting Rover in Three Languages

Jack (boy, age 12) is the only child to date to have significant MOO experience before joining MOOSE Crossing. A transcript of a conversation in which I asked him to compare the two appears in Table 3.2. (Jack was aware that he was talking to the designer of MOOSE Crossing, and his comments must be seen in that light.)

3.2 The Design of the MOOSE Language

Work on the MOOSE Crossing project began in September of 1992. It was referred to as "The MUD for kids I'm working on," until I came up with name MOOSE in June 1993. The client implementation was begun in 1992, and the server/language in November 1993. Children first used MOOSE in October 1995, as part of the Media Lab's 10th birthday celebration.

The core of the MOOSE language was designed in a series of weekly meetings between myself and my advisor, Professor Mitchel Resnick. MIT undergraduates Albert Lin, Trevor Stricker, and Austina Vainius helped design many of the finer details in innumerable impromptu discussions at my white board. Pavel Curtis, Randy Farmer, and Brian Silverman

[2]With the Epistemology and Learning Group's strong roots in the Logo community, it was tempting to start programs "to tickle" rather than "on tickle". However, in general, in MUDs an object holds scripts for things that can be done to it. A bear's tickle script enables people to tickle the bear, not the bear to do the tickling! This is better conveyed by "on" rather than Logo's traditional "to".

contributed design ideas on visits to the lab, and via email. Jon Callas and Dean Tribble also contributed ideas via email.

```
Amy says 'so I wanted to ask you... now that you've been here a
   while, what do you think of coding in MOOSE versus MOO?'
Jack says, 'moose is sooooo much simpler!'
Jack says, 'easier and a lot more fun'
Amy says 'in what ways is it simpler?'
Jack says, 'well, instead of this.location.announce_all it is just
   'announce''
Amy nods
Jack says, 'and i can make my own scripts, not just copy lines of a
   help file and changing them'
Amy says, 'cool!'
Amy says, 'did you notice the difference in how property references
   work?'
Jack says, 'yes, I also like how the edit script buttons and help
   buttons are seperate, easy windows'
Amy nods and listens
Amy says, 'anything you don't like?'
Jack says, 'I don't like when i crash every time i get red letters in
   programing a script.'
Amy laughs.  "Yeah, that was awful!  But you know we fixed it,
   right?"
Amy says, 'the new version of MacMOOSE doesn't do that'
Jack says, 'wow! I will have to convert!'
Amy says, 'definitely!'
Amy says, 'anything else you don't like?'
Amy says, 'or that you would change?'
Jack says, 'say where do i find the new version? hmm... somthig else
   i dont like...'
Jack says, 'i can't think of anything else!'
Amy says, 'the new version is on my web page where you got the first
   version'
Amy smiles
Amy says, 'http://asb.www.media.mit.edu/people/asb/'
Amy says, 'Does anything other than the software seem different to
   you?'
Jack says, 'hmm, i have been to many many muds, and moos, the only
   one that is close to comparing is mny old moo, du. But this is
   diffrent then that because of the @ commands...'
Jack says, 'and the fact that is is for kids'
Jack says, 'is is = it is'
Amy nods
Amy says, ‘do the people seem any different?’
Jack says, 'more friendly, i seem to interact with them better...'
```

Table 3.2: Jack’s Opinion of MOOSE versus MOO

MOOSE is built on top of MOO, and many aspects of its design are reacting to MOO, modifying it based on experience with the design of Logo and StarLogo in particular. Several principles emerged in the process of designing the language (See Table 3.3). I’ll discuss each in turn.

Design Principles:
- Have a gently-sloping learning curve.
- Prefer intuitive simplicity over formal elegance.
- Be forgiving.
- Leverage natural-language knowledge.
- Avoid non-alphanumeric characters wherever possible.
- Make essential information visible and easily changeable.
- It's OK to have limited functionality.
- Hide nasty things under the bed.

Table 3.3: Design Principles for the MOOSE Language

Parts of this chapter critique the design of MOO in detail. It should be noted that MOO is a production-quality, full programming language. MOOSE is quirky and incomplete. It continues to be extended and modified based on feedback from children. MOOSE is also roughly an order of magnitude slower than MOO. My goal in this critique is to bring forward design issues of broader interest—not to be critical of MOO.

3.2.1 A Gently-Sloping Learning Curve

For a programming language to be usable by young children, simple things need to be simple. In the earliest design meetings about the MOOSE language, we focused on how to make the learning curve as gentle as possible. Our first design decision, and one from which many other design decisions followed, was to make the programming language and the command line language as similar as possible. That way, kids can try a command out and put it in their program if it works. In MOO, the command line and programming languages are quite different. For example, in both MOO and MOOSE, to take an action (like smile or wink), you use "emote". If I type "emote laughs" everyone in the room sees "Amy laughs". Emote is typically one of the first commands that people on MUDs learn. In MOOSE, you can use that same command-line command in your program. Objects as well as people can emote. For example, Chester does this on his "computer" object. Chester is seven years old, and one of the youngest children to use MOOSE Crossing to date. When you type "eat computer" it tells everyone in the room "computer gobbels up a disk happoly." Here is the program:

```
on eat this
  emote " gobbels up a disk happoly."
end
```

In MOO, this same program would be:

```
this.location:announce_all(this.name + " gobbels up a disk
   happoly.");
```

In MOOSE, children can learn commands to use in conversation and then use those same commands in their programs. In MOO, an entirely new set of commands with different syntax needs to be learned for even simple programs.

Making the programming and command line languages the same helps more advanced MOOSE programmers as well as novices. For example, a child learning to use lists can try out list manipulation commands at the command line first seeing exactly how they work before trying to make them function in the context of a longer program.

The benefits of giving programmers immediate feedback can be traced back to the first interactive programming language, JOSS (Baker 1981). JOSS is cited as an influence by the developers of most early languages targeted at non-experts, including Logo (Feurzeig 1996), Smalltalk (Kay 1996), and BASIC (Kurtz 1981). There is no concept of a "command line language" distinct from a "programming language" in most interactive languages. The distinction was added in some MUD languages like MOO and LPC (the C-like language in which LPMUDs are constructed) to make a distinction between metaphorically acting in the virtual world and programming it.[3] This allows taking actions to be more natural-language-like, while programming looks more like typical programming languages. However, it adds a significant hurdle for people trying to learn to program, essentially removing the benefits of having an interactive language.

The similarity between the command line and the programming language is just one example of the many ways we tried to make the slope of the learning curve gentle. This principle was central throughout the process of language design.

3.2.2 Intuitive Simplicity Versus Formal Elegance

The second and perhaps most important design decision was to favor intuitive simplicity over formal elegance. In "Boxer: A Reconstructable Computational Medium," Andy diSessa comments that to create a popular medium, "a computer scientist's or a mathematician's measures of simplicity are simply not an issue. A better criterion is accessibility to a seven-year-old child" (diSessa and Abelson 1986).

---

[3]In the MUSE language, the command line and programming languages are identical. However, the integration of support for their distinct requirements is awkward. For example, to give something to someone in MUSE, you type "give person=object." In MOO or MOOSE, you "give object to person." More sophisticated parsing in MOOSE makes a smoother integration of command-line and programming language requirements possible. However, there is a trade-off: MOOSE's pattern-matching algorithm makes it substantially slower than MUSE.

There are compelling arguments in favor of paying attention to formal elegance. The idea is that through mastering some perhaps initially problematic construct, students will gain access to a way of thinking that will deepen their understanding in the long run. However, the problem arises: what happens if children never master the construct? Furthermore, does that transfer to broader concepts really occur, and are the broader concepts people are striving for really the right ones to be concerned about? What is this "formal elegance"? I agree with diSessa that the notion of elegance is a byproduct of an adult aesthetic, and largely irrelevant to children's needs.

The design of a language is primarily about tradeoffs. Given a choice between elegance and intuitiveness, we resolved to lean as far as possible towards intuitiveness. If you can have both, that's preferable; however, sometimes it's necessary to chose. Consider the following difference between Logo and StarLogo. Table 3.4 shows how you define a variable foo, set its value to 3, and retrieve the value of that variable in Logo, StarLogo, and MOOSE.

Language	**Set Variable**	**Access Value**
Logo	`make "foo 3`	`:foo`
StarLogo	`set foo 3`	`foo`
MOOSE	`set foo to 3`	`foo`

Table 3.4: Variables in Three Languages

In Logo, the quote before the variable definition makes it clear that foo is a symbol, to which we are assigning a value. The colon indicates that we are accessing the value of the variable and not merely referring to the symbol. The semantics are very clear, and highlight some underlying computer-science concepts. However, in years of working with children with Logo, Mitchel Resnick has found that the quote and colon are some of the most common causes of difficulty and error for children. It's not clear whether the broader concepts about symbols and quoting are commonly learned, but it is clear that these syntactic requirements slow down children's progress on their Logo projects (Resnick 1993).

In the StarLogo implementation , these concepts are a bit blurred. The technical explanation is that "set" is a special form that doesn't require quoting of its first argument. On declaring a new variable, an accessor function by the same name is automatically created which returns the value of that variable. This technical explanation is almost certainly lost on children using the language—they don't think about things like what a symbol is because there is no need to. There is less special syntax to learn and fewer things that can easily be gotten wrong (Resnick 1997).

The MOOSE version is almost identical to the StarLogo version. The added preposition "to" is consistent with MOOSE's natural-language-like style. This has the advantage that it draws on children's existing natural-language knowledge. It has the disadvantage that it may mislead them that more English-like commands work than actually do.

Which of these styles is preferable? In theory, you can make a case for any of them. However, working with children in practice, it seems clear that the Logo style is idealistically naive. The StarLogo and MOOSE approaches let children accomplish more. The children are better able to express their ideas, and therefore create more complex final products.

A cynic might ask, "If we're going to make things as simple as possible for the children, how about giving them an expert system that would create the entire project for them? Isn't the real educational value in doing the hard work of understanding some not immediately intuitive concepts?" To this I would reply, the educational benefit is not in the sophistication of the product; it is in the ability to express and refine a complex idea. If the child gets stalled early on by syntactic difficulties, the entire learning experience can be stalled. If the child is able to express ideas fluently, the child's learning experience will progress as the complexity of his or her project progresses.

These same issues emerged in the design of Microworlds Logo. In 1991, Logo Computer Systems International designed a new version of the Logo programming language called Microworlds Logo, and members of the Media Lab's Epistemology and Learning Group met several times to critique its evolving design. A heated debate ensued at MIT and in the broader community about the inclusion of paint tools. When most people think of children programming in Logo, they think of children learning geometry through drawing simple shapes. To make a square of size 100, you'd need to type: "repeat 4 [fd 100 rt 90]". The child's desire to make a picture provides a context that gives certain geometrical concepts new relevance. In Microworlds Logo, the child can simply select a paint tool from the menubar to draw a square or rectangle. Purists argued that this was robbing Logo of powerful potential for learning. If Logo were to be transformed into a simple paint program that would indeed be the case. Paint programs generally afford artistic learning experiences but not mathematical ones.[4] However, Microworlds is not merely a paint program—it is still the Logo programming language, and compared to other versions of Logo adds a series of features (buttons, fields, multiple turtles, multiple changing turtle shapes, etc.) that afford whole new categories of projects: simple animation and video games.

---

[4]Noteable exceptions are Mike Eisenberg's SchemePaint system (Eisenberg 1995) and David Shaffer's work with Escher's World (Shaffer 1996).

Paula Hooper worked with children at Paige Academy in Roxbury, Massachusetts for six years. For the first two years, her students used LogoWriter, a traditional version of Logo that works on Apple IIgs computers. The school then got Macintosh computers. For just a few weeks, the computers weren't equipped with Logo, but did have copies of Kidpix, a drawing program made by Broderbund Software. The students became very involved with making pictures in Kidpix. Soon after, the school received copies of Microworlds Logo. At first, Hooper had difficulty getting the children to stop drawing pictures in Kidpix and start working with Microworlds. However, she reports that once they realized what they could accomplish with Microworlds Logo, the children chose it over the paint program. Hooper worked with the same children using Microworlds for the following four years. The Paige Academy students are allowed to work on projects of their own choosing. Most use Microworlds to tell animated stories, and make their own video games. They have become enthusiastically involved in making much more complex projects than they were ever able to make in LogoWriter. This is of course partly because they are now older; however, Hooper also believes the affordances of the tool made much more exciting projects possible for them. They are learning less about geometry, but more about computational ideas (like procedural abstraction), and most importantly more about the process of designing and following through on a large, complex project. In LogoWriter, Hooper notes that even the simplest project was a substantial undertaking. The formalistic aesthetic that children should be forced to make squares out of combinations of ninety-degree turns is misguided. While some simple things (like making circles and squares) are done automatically for you in Microworlds, this makes more complex projects possible. The richest learning occurs through the process of designing a complex project that the student finds personally compelling (Hooper 1997a; Hooper 1997b).

An example of a trade off between formal elegance and intuitive simplicity in MOOSE is the question of quoting strings. In a MUD, it's natural not to quote most commands at the command line. You want to type:

```
say Hi there!  How is your project coming?
```

Not:

```
say “Hi there!  How is your project coming?”
```

However, we wanted to permit a more complicated syntax at the command line than is normally allowed in MUDs:

```
say “Today’s secret word is ” + my secret_word
```

You wouldn't normally use property references like that in conversation, but you might want to use them in programs. Therefore, you should be able to try them out at the command line.

Prohibiting unquoted strings at the command line would enforce a neater style, and help children to understand the notion of quoting. If we made that decision, the concepts would be clear: words always go in quotes; commands and variables don't. That would be the most elegant solution. But frankly, it's a hassle—no one really has spontaneous text-based conversations that way. And do you really want to start giving kids error messages every time they forget a quote?

The opposite extreme, never requiring quotes, causes implementation problems. In particular, we chose to make certain English words behave as operators in MOOSE, for example: *is*, *are*, *and*, *or*. How could you tell whether someone wanted to test equivalence or just use the word "is"?

We arrived at a complex compromise. At the command line, commands can be in one of two styles:

```
say In style one, the entire string is unquoted.  Since the strings
   around the operators are not quoted, the parser concludes that
   this must be an unquoted string, and leaves it alone.

say "In style two, the strings are quoted, and " + my adverb + "
   combined together."
```

In the first example at the command line, if you forget a plus or a quote, it is no longer a well-formed compound expression, and the system will treat the entire expression as an unquoted string.

In programs, you can get away with having unquoted strings, if they contain no infix operators. If they do, you will get a warning message when you compile them. An example appears in Figure 3.1. Feedback warning you about the problem appears in red in the feedback pane of the script editor window, immediately under your code. The system's ability to warn the user of potential problems helps avoid the problem of misunderstandings common to "do what I mean" (DWIM) systems.

The details of this solution may sound complicated, but to the user it amounts quite simply to: It's better to use quotes, but you can often get away without them.

In use to date with children, on the whole it is the younger children (nine years old and younger) who tend not to use quotes in programs. The decision to allow sloppy quoting has made MOOSE accessible to a wider range of ages and ability levels. The fact that the older children (ages ten to thirteen) tend

to quote their strings supports the fact that this sloppiness has led to no fundamental conceptual confusion.

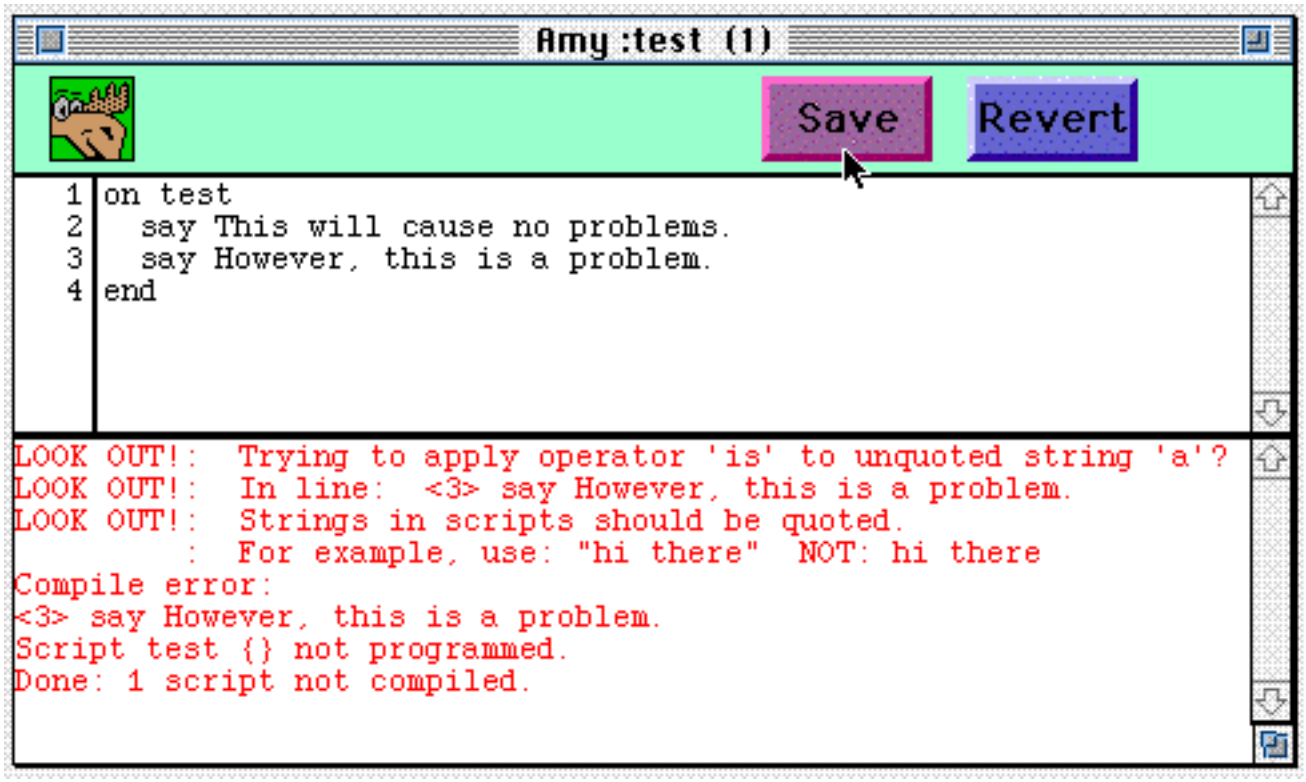

Figure 3.1: Unquoted Strings in a MOOSE Program

After reading a draft of this chapter, Danny Bobrow suggested that it would be desirable to have the system volunteer to correct the program. Instead of simply allowing quoted strings, the client could prompt the user asking whether they should be quoted. This might have significant pedagogical advantages. Making this approach successful would require the suggestions to have a high level of accuracy, and never to be repeated if rejected. Otherwise, the system's active interventions could be annoying. Bobrow's proposal merits further exploration in future systems.

3.2.3 Be Forgiving

MOOSE's tolerance of unquoted strings illustrates another important design principle: be forgiving. It's very often possible to anticipate certain common user errors and adopt a "do what they meant, not what they said" attitude. Being forgiving about quoting has made MOOSE accessible to younger children than would otherwise have been possible. In work with kids using MOOSE to date, we've found that they often begin by not quoting strings, but learn to do so as time goes on. Allowing unquoted strings in programs helps them complete their initial projects more easily. Success in those initial projects helps to deepen their long-term involvement.

The disadvantage of this approach is that it sometimes makes it difficult to give the programmer helpful error messages. For example, consider this program:

```
set result to 3
say resilt
```

This MOOSE program will say the word "resilt" (note the typo) instead of saying 3, the value of the variable "result". In a language that doesn't allow unquoted strings, the second line could generate an error message because "resilt" is undefined. MOOSE assumes "resilt" is an unquoted string. However, MOOSE is able to give somewhat better feedback to this error:

```
set result to 3
say "The answer is " + resilt
```

Compiling this program gives this feedback:

```
LOOK OUT!:  Trying to apply operator '+' to unquoted string 'resilt'?
LOOK OUT!:  In line:  <3> say "The answer is " + resilt
LOOK OUT!:  Strings in scripts should be quoted.
         :  For example, use: "hi there"  NOT: hi there
Script 'typo_example' programmed.
Done: 1 script compiled.
```

An unquoted string on its own is undetectable; however, applying an operator to an unquoted string generates a warning message. The error message is not ideal, but at least it alerts the user to the correct location of the problem. The "do what they meant" design philosophy allows us to give good feedback most but not all of the time.

3.2.4 Leveraging Natural-Language Knowledge

One potentially dangerous decision we made was to leverage off of children's natural language knowledge. For example, in MOO, property references are of the form `<object reference>.<property name>`. MOOSE property references use an English-like possessive notation. The possessive notation is natural and children pick it up without having it explained to them. Their natural language knowledge makes the use of property references easy.

MOO:	<object reference>.<property name>
Examples:	#99.age
	#99.owner.name
MOOSE:	<my/object's/objects'> <property name>
Examples:	my name
	Ello's name
	Ello's owner's description
	Snuggles' description

Table 3.5: Property References in MOO and MOOSE

The first language to use English-like syntax in order to be more accessible to non-professionals was COBOL. Jean Sammet, one of the designers of COBOL, writes:

> Although from the very beginning COBOL was concerned with "business data processing," there was never any real definition of that phrase. It was certainly intended (and expected) that the language could be used by novice programmers and read by management. We felt the readability by management could and would be achieved because of the intended use of English, which was a fundamental conclusion from the May 1959 Pentagon meeting. Surprisingly, although we wanted the language to be easy to use, particularly for nonprofessional programmers, we did not really give much thought to ensuring that the language would be easy to learn; most of our concentration was on making it "easy to read" although we never provided any criteria or tests for readability. (Sammet 1981)

In the design of MOOSE, we put as much emphasis on "learnability" and "writeability" as in "readability."

More than twenty-five years after the design of COBOL, the designers of Hypertalk had similar goals and strategies. When asked about the language ancestors of Hypertalk, designer Bill Atkinson replied "The first one is English. I really tried to make it English-like" (Goodman 1988). Ted Kaehler, another member of the Hypertalk design team, comments that "One principle was 'reads as English, but does not write as English.' Like an ordinary programming language, it depends on exactly the right syntax and terms" (Kaehler 1996).

In designing MOOSE's natural-language-like syntax, we drew most heavily on MOO's command-line language (Curtis 1993), but also borrowed directly from Hypertalk. For example, MOOSE borrows Hyper Talk's use of the variable "it" to refer to the last value returned.

The risk of making a computer language like a natural language is that people will assume more natural language constructs work than really do. On the whole, this has not proved to be a problem. MOOSE commands have a consistent syntax: each command begins with a verb. The arguments a command takes are readily viewable with system commands like "examine," "which," and "show."

The slippery slope of natural language has proved to be a problem in only one area: conditionals. While the syntax of most commands roughly follows a simple "verb direct-object preposition indirect-object" pattern, the syntax of the Boolean clauses of if, elseif, and while statements is much more complicated. Most of the conditionals the kids have written so far have been simple enough to pose few problems. But consider these lines (from a script by me, the "teach" script on Generic Teachable Object):

```
if prop member_of my teach_locked and player is not my owner and not
   player's admin
  tell player "Permission denied."
  return
endif
```

This does indeed have a specific syntax: clauses are separated by the conjunctions "and" and "or"; individual clauses are usually of the form <argument> <operator> <argument>, or simply <argument>. However, this syntax is more complex than other MOOSE constructs; consequently, it's easier to start assuming arbitrary English statements will work.

The most common mistake we observed was for kids to put a "not" in the wrong place. For example, they might write "A is not B". This leads to some potential confusion as to whether the negative applies to the element B or to the clause. To clear this up, I made the compiler always assume a negative applies to the clause—they presumably meant "not (A is B)". Negating an individual element is an advanced concept that I saw no advantage in introducing. (It's still possible to do so—you just need to use parentheses: "A is (not B)".) The second most common mistake was to add an extra "is." For example, they might write "item is member of list" instead of "item member of list." The words "is" and "member" are separate operators, and their composition is nonsensical—the programmer clearly meant just to use the "member" operator. The easy fix for this is simply to eliminate any instances of "is" before another operator. This was easy to do in the compiler. Another simple fix was to allow certain operators like "member_of" be written as either one or two words. These technical improvements have greatly reduced the number of errors kids encounter. However, it still remains somewhat difficult to debug bad conditional expressions. This is an area in which the language could be improved.

Over all, the similarity of MOOSE to natural language has proved to be a good design decision. Kids often look at other kids' programs and understand them without having them explained. They are immediately readable. One of the most common learning techniques I've observed kids using is to start with a simple variation on another child's program. Later they progress to making increasingly original creations. This immediate understanding of programs is a result of the language's natural-language-like structure.

3.2.5 Avoid Non-Alphanumeric Characters

One of the easiest design decisions we made was to avoid non-alphanumeric characters wherever possible. Kids aren't familiar with them, and identifying them and typing them pose problems. In particular, many commands in MOO are preceded by the character @. The logic behind the @ command in MOO is this: commands to the programming environment use @s; commands that simulate taking actions in the virtual world do not. In practice this rapidly breaks down. Many commands are user-defined, and not every programmer understands or respects the convention. Furthermore, whether a command is equivalent to taking action in the virtual world is not always clear. For example, if you're reading a political ballot on LambdaMOO but the ballot object is not in the room but is defined system-wide, are you reading it or @reading it? What about if you're reading your MOO mail? The former is "read" and the latter is "@read". It's hard to keep them straight. MUSE also uses @s, but with a different rationale. In MUSE, commands that have a side effect to change something in the database are preceded by an @. This also is somewhat ambiguous. For example, walking around the virtual world does not require an @, yet that does change something in the database—your location. Teleporting on the other hand does require an @ in MUSE. The very fact that the convention for when to use an @ varies between languages also significantly contributes to the confusion. People frequently use multiple environments, and may not realize that the convention differs. We resolved this in MOOSE by eliminating @s altogether.

One common problem for novice programmers is the tendency to confuse using an equals sign for assignment and for equivalence. In MOOSE we avoided this problem by using "set" for assignment (i.e. "set my age to 30"), and English words for equivalence: is, are, isn't, and aren't. Other operators are also converted to words. MOO's operator "&&" becomes "and"; "||" becomes "or".

In addition to being unfamiliar and hard to type, non-alphanumeric characters look "high tech." Much of MOOSE is learned by looking at sample programs. Letters are familiar to children, and less threatening. Programs filled with special characters may tend to scare off new users by making coding look hard.

In a visit to a classroom of “Title I” students (students who are more than two years behind in their reading level) using MOOSE Crossing, I noticed that those students had particular difficulty typing unfamiliar characters. Characters that require the use of the shift key appeared to be particularly problematic. While I eliminated most special characters from MOOSE, a few are still used occasionally, particularly underscores. Observing the Title I students, it was clear that this was a mistake. They will be removed in the future. While they appeared to be struggling with simple reading and writing, the special characters seemed to be an even bigger hurdle.

English words have every-day-use meanings that often make it easier for them to understand and remember the computer meanings of those words. Special characters usually don’t have those ordinary meaning, so their computer meanings are more easily confusable. This became abundantly clear with one special character we did include in MOOSE, single-quote ('). The two most commonly used commands in MUDs are “say” and “emote.” They are used so often that it’s common in most MUDs to allow them to be abbreviated as double-quote (") and colon (:) respectively. In MOOSE, we wanted to avoid confusion between the abbreviation for say and quoted strings, so we made the abbreviation be single-quote ('). One common point of confusion for people new to MUDs is the distinction between saying something and doing it. Contrast the following three commands:

A.

```
--> say down
You say 'down'
```

B.

```
--> 'down
You say, 'down'
```

C.

```
--> down
You climb down the rope to the platform.
```

Saying the word "down" is not the same thing as trying to actually go down, changing your location in the virtual world. It's easy to see the different between A and C above. It's harder to see the difference between B and C. Here's an edited excerpt from a confusing afternoon when a class of twenty students connected to MOOSE Crossing for the first time. Experienced users Miranda (girl, age 10-11) , Rufus (boy, age 12), and Newton (teacher, male, age 41) came by to help the new users. Austina and I were also there. Not long after all twenty students connected for the first time, this conversation ensued:

```
Amy says, 'wow, quite a crowd!'
jj says, 'Who is Pumpernickle''
cj says, 'cj here'
```

```
Bill says, 'pumpernickel''
Miranda says, 'A dog'
Tim says, 'hi Tag'
Amy [to jj]: Pumpernickel is my dog. Try this: pet pump
Newton floats above the crowd, nervous and shy.
Rufus says, 'You guys can have dogs too.'
jj says, 'Can we pet him/her''
Miranda says, ''Yeah.'
Rufus says, 'It's up to Amy...'
Amy says, 'you can tickle her too'
Bill says, 'BeeBee''
jj says, 'pet Beebee''
Tag says, 'pet beebee'
isaac says, 'hi pet beebee''
Tim says, 'pet BeeBee''
Amy [to jj]: don't say it, do it. Don't put the ' in front
Bill says, 'pet BeeBee'''
```

BeeBee is their teacher's virtual pet dog. Presumably, the students' teacher is suggesting they pet BeeBee. This doesn't succeed for two reasons. First, they are saying the words instead of taking the action by accidentally prefacing their commands with a single quote. Second, BeeBee isn't in the same room in the virtual world as the students.

```
Rufus says, 'If anyone wants to see something cool, type 'enter
   Sparky'.'
jj says, 'pet BeeBee''
[Rufus types: enter Sparky
[From Sparky III] Duggan wags his tail at Rufus.
Bill says, 'Sparky''
isaac says, 'sparky''
Miranda pets Pumpernickel.
Pumpernickel wags her tail and licks Miranda's hand.
Jermaine says, 'Hi Lara''
Tim says, 'enter Sparkey'
Amy says, 'don't put a 'say' or a quote in front of the word pet'
Alana sniffs Rufus curiously.
cj says, 'Bill what class are you in''
jj says, 'enter Sparky''
Miranda pets Pumpernickel.
Pumpernickel wags her tail and licks Miranda's hand.
Lara says, 'pet beebee'
Rufus says, 'type 'enter Sparky''
Miranda says, 'This is too confusing!'
Bill says, 'enter Sparky''
jj says, 'enter Sparky''
isaac says, 'sparky''
Austina goes home.
Alana follows after Austina.
Amy [to Bill]: don't put the ' or say in front of it. Type it just
   like this: enter sparky
Rufus says, 'No! Type it on the keyboard!'
isaac says, 'look''
jj says, 'look''
Tag says, 'look''
Lara says, 'look''
Tim says, 'By austina''
```

```
Rufus says, 'Thank you Amy!'
Jermaine says, 'look''
```
[At this point, Bill gets it right and enters Sparky]
```
[From Sparky III] Duggan sniffs Bill curiously.
Bill You climb onto the main deck.
```
[5]

A number of things contributed to making this encounter confusing. The sheer number of students was a significant factor. Their teacher had only tried MOOSE Crossing a couple times herself. In fact, she was giving the class incorrect instructions out loud, telling them to type commands both starting with and ending with single quotes. (This led to the extra single quote at the end of most things the new members said.) She told them to do this for all commands, not just things they wanted to say. Once this misconception was created, it was hard to correct, particularly since single quotes are used to delimit ordinary conversation. When you give someone instructions on what to do, single quotes wrap what you are saying. It therefore is difficult to talk about when to use single quotes. The confusion persisted for some of the students even when they returned for a second session the following week. Perhaps the most important contributor to the confusion was the design decision to allow the use of single-quote as a shortcut for the "say" command. The teacher was giving the students explicit wrong directions; however, the teacher herself might not have been confused if the command had been clearer. Furthermore, even if the teacher had given them incorrect instructions, the students would have been more likely to figure out what was wrong if they had been using the English word "say" instead of an ambiguous punctuation mark. All the students were using single quote, not "say," in this conversation. It's easy to understand how they came to type these incorrect commands:

```
'pet BeeBee'
'look'
'enter Sparky'
```

It's harder to imagine them making the equivalent mistakes if they knew only the "say" form:

```
say pet BeeBee
say look
say enter Sparky
```

New MOOSE Crossing members usually learn to use "say" first, and only learn the single quote shortcut later. This significantly reduces the potential for confusion. This level of confusion is not typical. However, it still reinforces the broader point: non-alphanumeric characters are best avoided if possible.

---

[5]Several lines were edited out of this transcript for clarity and conciseness.

3.2.6 Make Essential Info Visible and Easily Changeable

In some programming languages, for example C and MOO, information about a function is stored separately from the code for the function itself. This separation is confusing. While trying to fix a problem, users often find that the information needed to fix the problem is not in front of them. Remembering to check the information about the function (or property) as well as its contents is a debugging strategy that most new programmers are slow to master. It's advantageous to keep essential information with the code itself.

In particular, both MOO verbs and MOOSE scripts have arguments. In MOO, these must be declared when the verb is declared:

```
--> @verb me:jump this on any
Verb added (8).

--> @verb me:jump this none this
Warning:  Verb `jump' already defined on that object.
Verb added (9).

--> @verbs me
;verbs(#75) => {"enlist", "random_player", "hack", "words", "jump",
   "jump"}
```

MOO argument specifications must be three items long (direct object, preposition, indirect object), and the preposition choices are drawn from a fixed set of alternatives (Curtis 1993). The special argument specifier "this none this" is defined as meaning that the verb is not intended to be called from the command line, but only from other verbs. Working with multiple verbs of the same name is extremely difficult, because the interface has few affordances for indicating which verb of a given name you mean. Listing the code for a verb, you do not see its arguments. To see its arguments, you must use a special command like "@show" or "@display". To change the arguments, you must use another special function: "@args"; you can't use the same mechanisms you use to change the code.

In MOOSE, argument specifications can be of any length, and any word may be used as a constant. A script's argument declaration is simply the first line of the program. It's clearly visible as you edit the program, and can be changed by the same mechanism you use to edit the program. Multiple scripts with the same name can simply be placed one after the other in the script body, with no confusion between them. For example, here's the "feed" script on Generic Penguin by Rachael (girl, age 12-14):

```
on feed this
  tell context "You feed " + my name + " fish."
  announce_all_but context context's name + " feeds " + my name + "."
  set my ishungry to 1
  emote "glups the fish down hungrily."
  set my last_feed_time to time
end

on feed this string
  tell context "You feed " + my name + " a " + string + "."
  announce_all_but context context's name + " feeds " + my name + " a
   " + string + "."
  set my ishungry to 1
  announce_all my name + " glups the " + string + " down hungrily."
  set my last_feed_time to time
end

on feed this herring
  tell context "You feed " + my name + " pickled herring."
  announce_all_but context context's name + " feeds " + my name + "
   pickled herring."
  emote "puckers up and spits the herring out."
  emote "blahs!"
end

on feed this shrimp
  if my diet is 1
    tell context "You can't feed " + my name + " shrimp today because
   " + my name + " is on a diet."
    announce_all_but context context's name + " tries to feed shrimp
   to " + my name + " but " + my name " is on a strict diet!"
  else
    set my ishungry to 1
    tell context "You feed " + my name + " a ton of shrimp!"
    announce_all_but context context's name + " feeds " + my name + "
   a load of shrimp."
    emote "gobbles the shrimp down in seconds."
    set my last_feed_time to time
  endif
end
```

```
on feed this eggs on toast
  set my ishungry to 1
  tell context "You feed " + my name + " a soft boiled egg on toast."
  announce_all_but context context's name + " feeds " + my name + " a
   soft boiled egg on buttered toast."
  emote "politely picks up the toast with " + my name + "'s left
   flipper and nibbles the end like so."
  announce_all_but context "After chewing, " + my name + " thanks " +
   context's name + "."
  tell context my name + " says, 'Thank you ever so much in offering
   me this delightful meal. I would be ever so obliged if you would
   be so kind as to give me a cup of tea to wash it down.'"
  set my last_feed_time to time
  set my askfortea to 1
  fork 10
    if my askfortea is 1
      tell context my name + " says, 'Thanks alot!'. (You didn't give
   " + my name + " any tea!)"
      set my askfortea to 0
    endif
  endfork
end

on feed this tea
  if my askfortea is 1
    tell context "You give " + my name + " a cup of tea."
    set my askfortea to 0
    announce_all_but context context's name + " gives " + my name + "
   a cup of tea."
    emote " picks up the cup from the saucer, and takes a sip."
    tell context my name + " says, 'Thank you ever so much for the
   tea. I feel honored that you decided to honor my request for
   tea.'"
  else
    tell context "You give " + my name + " a cup of tea."
    announce_all_but context context's name + " gives " + my name + "
   a cup of tea."
    emote "slurps up the tea."
    emote "nods in thanks."
  endif
end
```

Andy diSessa argues that programming languages should be "tuned toward small tasks—the ability to implement simple ideas easily is much more important than the ability to do complex tasks efficiently"(diSessa and Abelson 1986). The ability to have lots of short scripts of the same name helps break tasks into parts more clearly.

3.2.7 It's OK to have Limited Functionality

In our first serious discussion of the design of MOOSE, Pavel Curtis looked at me squarely and said: "If you can do everything in MOOSE that you can in MOO, I'll be disappointed in you." The designers of Logo pride themselves on the fact that Logo is a full programming language that you could use for professional software design if you wanted. The ideal is that a tool should

have no initial barrier and no ceiling—no limits on what a kid can achieve. While this is an inspiring vision, the truth is that few kids actually reach those higher levels. If advanced functionality comes at no cost, it's certainly desirable. However, in the design of MOOSE when we encountered a trade-off between supporting advanced features and making common features simple, we always tried to give priority to simplicity. MOOSE is tuned to make the set of things that are natural to do in a MUD—like bears that you can tickle—as easy as possible.

3.2.8 Hide Nasty Things Under the Bed

Perhaps one of the most-discussed issues in programming language design is how much detail to hide from the user. Maximum efficiency dictates that the user take direct control of as many functions as possible. For example, C requires users to allocate and deallocate memory manually. Usability usually dictates that the system do as much as possible of the unpleasant and error-prone parts of the task. For example, Lisp and MOO do automatic garbage collection. It should come as no surprise that for MOOSE, we chose the second approach, hiding difficult concepts where possible. In particular, we succeeded in hiding issues of permissions, atomicity and multitasking.

Permissions

Security for multi-user environments are an important topic of current computer science research. Over time, through intensive use and frequent attacks, the LambdaMOO server and LambdaCore[6] database have been made robustly secure. The permissions system that has evolved, however, is the most difficult part of MOO to master.

One particularly thorny issue concerns ownership of properties. Objects have owners, and individual properties also have owners. Properties have permission bits r (readable), w (writeable, which is almost never used), and c (change owner on inheritance). Suppose that Ginny owns Generic Dog, and Amy has a child of Generic Dog named Pumpernickel. Generic Dog has a bark_msg property. Now suppose Generic Dog has a "guard" script that changes the dog's bark_msg from a polite yip to a deep growl. If the bark_msg property is set +c (i.e. with the c bit set), then Amy will own Pumpernickel's bark_msg property, and can change it directly. However, the "guard" script runs with Ginny's permissions, so now it can't change the dog's bark_msg. If the property is !c (i.e. with the c bit unset), then the "guard" script can change the dog's bark, but Amy can't. The solution is to make it !c, and have Ginny write a change_bark accessor script allowing dog owners to change their dogs'

[6]To start a MOO, you need both the server and an initial database. It's possible to start with a minimal database that adds little or no functionality. However, most MOOs use a database extracted from LambdaMOO (the first and still most popular MOO) called LambdaCore. Among the features included in LambdaCore is the line editor discussed here.

barks. If you're confused, that's the point—permissions are difficult to understand.

In MOOSE, I found a way around this problem. I added a trust_parents property to all objects. If an object trusts its parents, then any code on an object's parents may modify any of its properties. All MOOSE properties are +c—if you own the object, you own the property. The concept of the c bit is eliminated. Generic_Dog's growl script can modify Pumpernickel's bark_msg because Pumpernickel trusts her parents. (This does not mean that other code by Ginny can modify anything about Pumpernickel—just code on the parent object.) This involves assuming some trust between the owner of a generic and the owner of the child objects. However, that trust already exists. The generic owner can already add any property or script he or she likes to the parent, which will be inherited by all child objects. The generic owner would also be able to modify properties in the common MOO solution to this problem described above. MOOSE lets you do the things that it seems natural to do—have both Generic Dog and Pumpernickel's owner be able to modify Pumpernickel's bark_msg. It's not necessary for the kids to think about property permissions at all.

The w (writeable) bit is not supported either. There is no good reason in MOOSE or MOO to make anything world-writeable. The r (readable) bit for properties is a more difficult issue. It's beneficial to the community if scripts and properties can be used as examples by others in the community. On the other hand, making things unreadable is useful for applications like secret passwords, which children often are interested in writing. I was initially hesitant to introduce the notion of a property permission at all. However, after some discussion between Austina Vainius and myself and the independent suggestion of the same idea by Pavel Curtis, we finally compromised on adding the notion of a "hidden" property. We called them "hidden" rather than "secret" to make them sound less exciting. It's desirable to minimize their use, so that people can learn from one another's projects as much as possible.

Atomicity and Multitasking

In any time-sharing computing system, you need a system for deciding how to share processor time. MOO uses single-threaded, non-preemptive multitasking. Each task has a limit on the number of ticks and seconds it can use. If it runs out, the task stops with an out-of-ticks or out-of-seconds error. Before a task runs out of time, the programmer can voluntarily give up control using the "suspend" command. When the task returns to the top of the queue, it has refreshed tick and seconds counts (half as many as it originally started with).

I wasn't particularly in the mood to teach children about tick limits and suspending. Furthermore, MOOSE adds sufficient overhead to the server

that even simple tasks often must suspend at least once before completing. I decided to make MOOSE automatically suspend when needed.

This model has drawbacks. If you are petting a dog, you can no longer assume that a dog that was in the room at line 1 is still there at line 10! There is no way to guarantee that any task completes atomically. In practice, this has not yet caused significant problems. A few kids have noticed that messages generated by objects entering a room sometimes arrive in an odd order. For example, my dog Pumpernickel wags her tail when she enters a room containing someone who has pet her in the past. The message that she wags her tail often precedes the message notifying you that she has arrived in the room. Each command in an individual program is guaranteed to be executed in the correct order; however, the arrival message and the tail wagging are generated by different programs on different objects. The order of turn-taking between programs running on different objects is not guaranteed. While this simple example can be passed off as an oddity, the problems are likely to become more severe as the system grows in complexity.

As Mitchel Resnick's research on StarLogo has shown, most people are not generally comfortable thinking about complex, parallel systems. StarLogo seeks to make that complexity understandable and interesting in itself (Resnick 1994). I chose not to make understanding these issues a pedagogical goal for MOOSE. Instead, I decided to hide them from the user as much as possible. This works for limited applications, and has helped make the system accessible to novice programmers. However, this simplistic solution has limits. More research is needed to devise a gracefully scaleable solution to meet the needs of both novices and experts, and to enable the construction of more complex systems.

3.2.9 A Design Philosophy

There are few "right" answers in programming language design—primarily, there are trade-offs. Furthermore, it's not particularly meaningful to talk about one language being "better" than another. However, it is meaningful to talk about the advantages of a language for a particular group of people with a particular set of goals. And languages do have affordances—things they make easy to express, and things they make hard to express. Larry Wall, inventor of the Perl language, compares different programming languages to different styles of music—C is reductive and rigid like the Modernism of John Cage; "C++ is like movie music, of titanic proportions, yet still culturally derivative"(Wall 1996). Wall argues that programmers are like artists, and different languages are like different types of materials—they are expressive in different ways, and suited to people with different goals and personalities.

Our goal in designing the MOOSE language was to give children a new expressive medium. Towards this end, we tried to make it as easy as possible for children to write the sort of programs one tends to want to write in a

MUD. Our general approach was deliberately a bit mischievous, exploring a design aesthetic that is counter-intuitive for most computer scientists: put simplicity and immediate intuitiveness first, and ignore as much as possible many of the concerns that adult computer scientists tend to value. We tried to ground the design in experience working with real children with Logo, and revised the design based on feedback from initial users. Detailed analysis of what children have been able to accomplish with MOOSE forms the subject of much of the rest of this thesis.

3.3 The Need for a New Programming Environment

While existing MUD languages present a barrier to children learning to program, available programming environments are an even bigger problem. In MUSE, every command is one-line long, so no editor is necessary. Programming in MOO requires the use of an editor. The "@program" command used to enter programs is built into the server software. To modify your program using just that command, you would need to type the entire thing again. A very nice Emacs-based editor, mud.el, is available. Mud.el gives you an excellent interface to edit both MOO verbs and properties. However, not everyone has access to Emacs, and it's certainly not appropriate for children. An alternative, currently used by the Pueblo project, is the LambdaCore line editor.

Here is the help for editor commands:

```
Verb Editor
Commands:

say         <text>                      w*hat
emote       <text>                      e*dit       <obj>:<verb>
lis*t       [<range>] [nonum]           com*pile    [as <obj>:<verb>]
ins*ert     [<ins>] ["<text>]           abort
n*ext,p*rev [n] ["<text>]               q*uit,done,pause
enter
del*ete     [<range>]
f*ind       /<str>[/[c][<range>]]
s*ubst      /<str1>/<str2>[/[g][c][<range>]]
m*ove,c*opy [<range>] to <ins>
join*l      [<range>]
fill        [<range>] [@<col>]

----  Do `help <cmdname>' for help with a given command.  ----

<ins> ::= $ (the end) | [^]n (above line n) | _n (below line n) |
   .(current)
<range> ::= <lin> | <lin>-<lin> | from <lin> | to <lin> | from <lin>
   to <lin>
<lin> ::= n | [n]$ (n from the end) | [n]_ (n before .) | [n]^ (n
   after .)
`help insert' and `help ranges' describe these in detail
```

Here's how the program to pet Rover (discussed in Section 3.1) would be created and entered using the LambdaCore line editor:

```
>@verb rover:pet this none none
Verb added (0).
>@edit rover:pet
Verb Editor
Do a 'look' to get the list of commands, or 'help' for assistance.

Now editing #2156:pet.
>enter
[Type lines of input; use `.' to end or `@abort' to abort the
   command.]
>player:tell("You pet Rover.");
>this.location:announce_all("Rover wags his tail.");
>.
Lines 1-2 added.
>compile
#2156:pet successfully compiled.
>q
Amy's Office
Amy's office is a jumble of books and papers.
```

Being able to break code up into multiple lines instead of cramming lines together as they are in MUSE is a significant improvement, but it comes at the price of adding this elaborate editor. To change "his" to "her" in MOO without a client, we would need to do this:

```
>@edit rover:pet
Verb Editor

Do a 'look' to get the list of commands, or 'help' for assistance.

Now editing #2156:pet.
>list
  1: player:tell("You pet Rover.");
__2_ this.location:announce_all("Rover wags his tail.");
^^^^
>s / his/ her/2[7]
__2_ this.location:announce_all("Rover wags her tail.");
>compile
#2156:pet successfully compiled.
>q
Amy's Office
Amy's office is a jumble of books and papers.
```

Perhaps the worst feature of the editor is that it actually moves you to a separate room to do your coding. The rationale for this design decision was to allow reuse of the "say" command for inserting text, and give you access to a simple set of editing commands. Moving you to a separate room was the

[7]Note the spaces before "his" and "her." The first time I typed this while writing this chapter, I actually forgot the space and ended up changing "this" to "ther." Even for an experienced user, this kind of editing is frustrating.

simplest way of achieving a form of modal interface (Curtis 1996). The downside of this design decision is clear. One of the great strengths of MUDs is their collaborative nature. Using the LambdaCore line editor, you are sent off to a room all alone whenever you try to work! (The collaborative nature of learning in MUDs will be discussed in more detail in Chapters 4 and 5.)

3.4 The Design of the MacMOOSE Client

There were few if any research issues involved in designing the MacMOOSE client program—just attention to detail, interface design work, and an iterative design process incorporating feedback from children. While I implemented the MOOSE language, code for the MacMOOSE client was written by several MIT undergraduates, participating in the project through MIT's Undergraduate Research Opportunities Program (UROP). The interface design was done by myself and the students jointly. The students who worked on the project are listed in Table 3.6.

Greg Hudson	September 1992—May 1993
Adam Skwersky	June 1993—May 1994
Steven Tamm	February 1994—May 1995, and September 1995—May 1996
Jon Heiner	February 1995—May 1996
Drew Samnick	January 1996—May 1997
Steven Shapiro (Java version)	September 1996—present

Table 3.6: The MacMOOSE Development Team

MacMOOSE allows you to:

- Edit properties and code, and send MOOmail in Macintosh (WYSIWYG) style,
- Open an "object browser" to see all the scripts, verbs, and properties on an object, and
- View help in a separate window.

The application was designed to make it as easy as possible to learn to program, and to do creative writing. Features that were suggested by users or members of the development team that were excluded as not contributing to this goal include for example: automatic mapping, a separate window displaying who is currently connected, and a separate window showing what you are holding. Suggested features that were not implemented due to lack of time but which would advance the project's primary goals include: flashing matching parentheses, automatically coloring keywords in code, search and replace, support for alternate character sets (for students using MacMOOSE in language classes), and hypertext help (allowing you to double-click on a word in order to get help on that topic). Much of the time spent in design meetings

was devoted to fighting "featuritis," in order both to keep the interface easy to use and to keep the development time manageable.

We chose to develop MacMOOSE in Symantec C, because that was the best available development platform in September 1992. Between MacMOOSE version 1.0b1 and version 2.0a1 we moved to Symantec C++, which enabled more code reuse and better design of class abstractions. There was no adequate platform-independent development environment available at the time the project was started. A significant disadvantage of using the MacMOOSE client program is that it requires a Macintosh with a direct Internet connection. Had Java existed when we began its development, we would have written it in Java to reach a broader variety of platforms. A Java version of MacMOOSE is currently under development. While Java runs on multiple platforms, those platforms all require direct Internet connections as well. The LambdaCore line editor will run on anything—even a dumb terminal connecting over a phone line. The greater functionality of MacMOOSE is achieved at the cost of platform dependence.

3.4.1 A Tour of MacMOOSE

Connecting to MacMOOSE, you first select a server from a list (Figure 3.2). You can have multiple connections open at once. Each connection is identified by a number, and you can switch between them using command keys. Windows associated with a particular connection are grouped together on the windows menu. (These features are more necessary for adult MOO programmers than for children using MOOSE. As well as supporting children programming on MOOSE Crossing, MacMOOSE also works as a general-purpose MOO client. As of June 1996, more than 950 people had registered copies.)

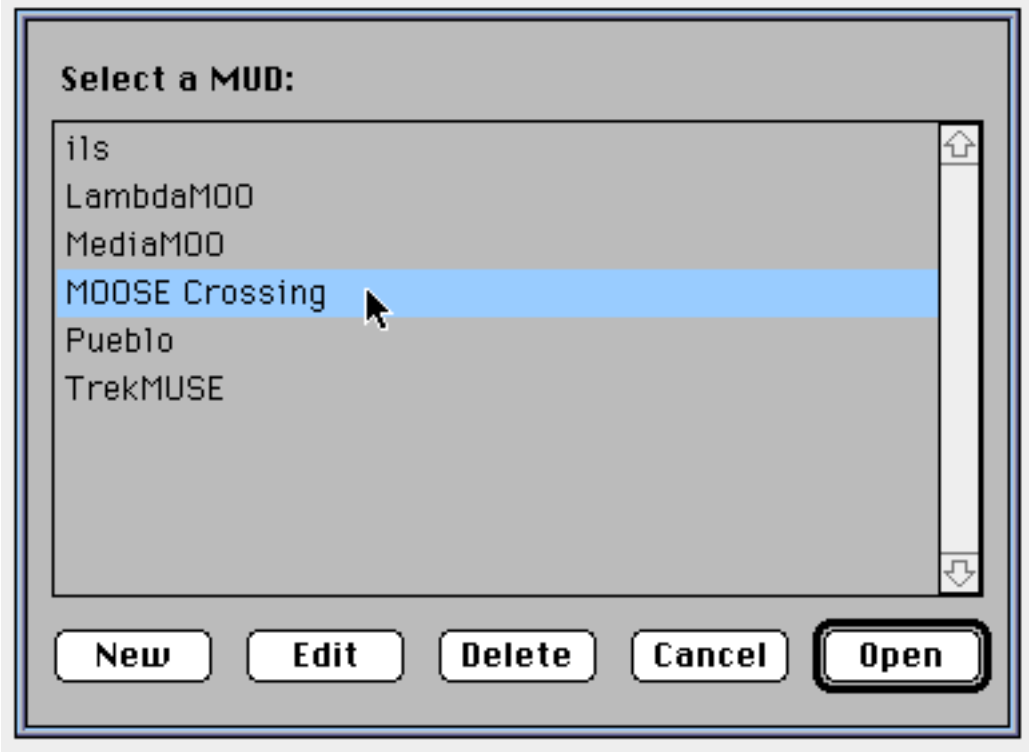

Figure 3.2: The MacMOOSE Server List

A password dialog lets you enter your password privately (Figure 3.3). With most other MUD clients, your password echoes in clear text. We worried this might lead to some children pulling pranks after they had seen each other's passwords.

Once connected, your main connection window is divided into two panes: input and output (Figure 3.4). In a raw telnet connection, incoming text can make it difficult to read the line you're in the middle of typing. The most common reason people use MUD clients is to solve this interface problem.

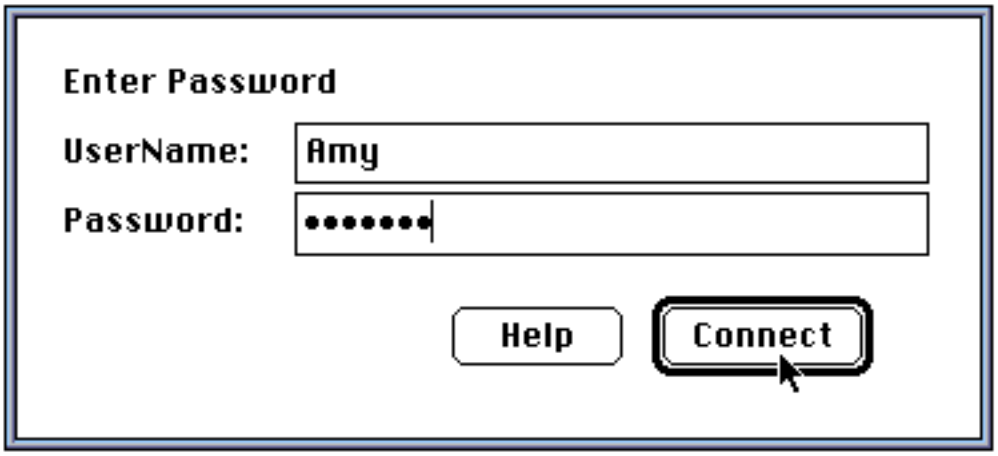

Figure 3.3: The MacMOOSE Password Dialog

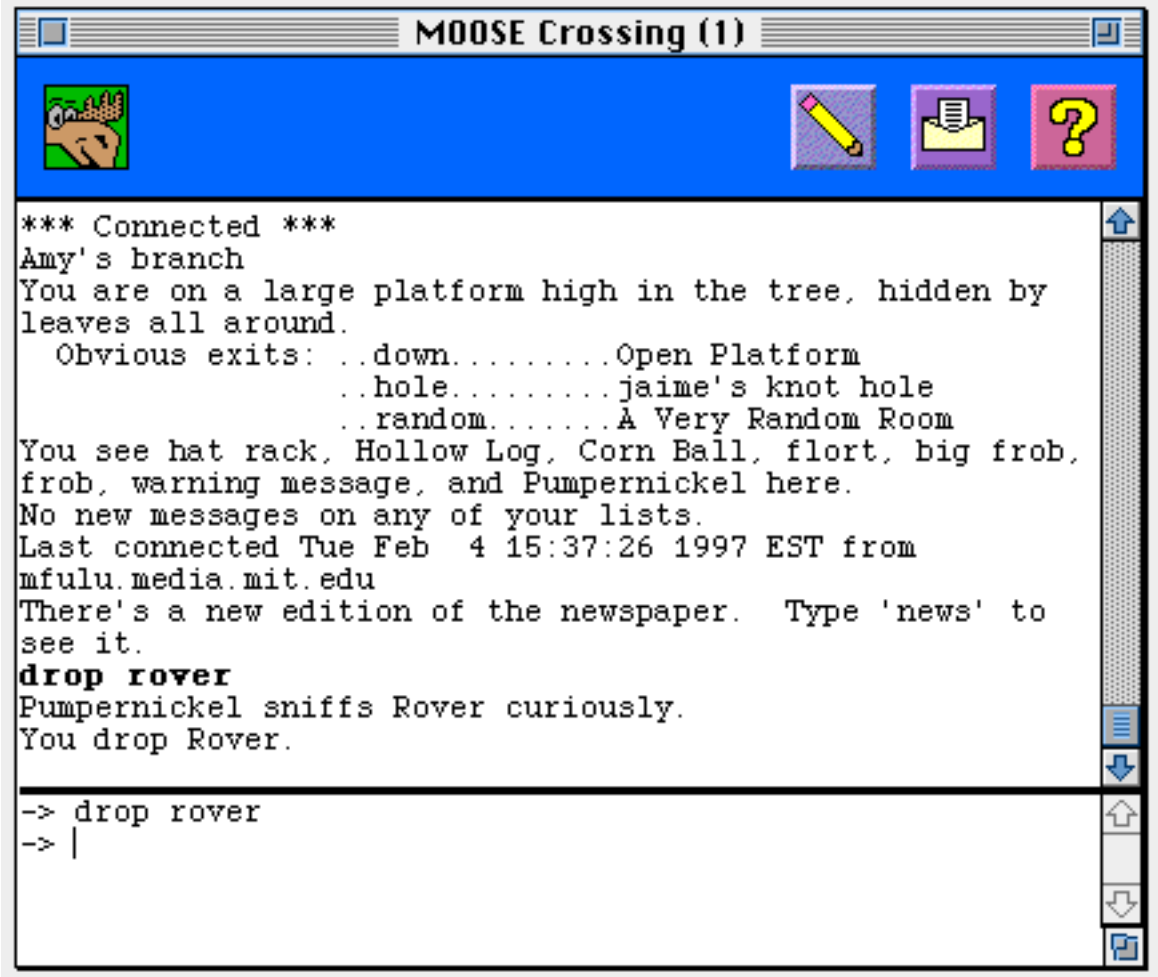

Figure 3.4: The MacMOOSE Main Connection Window

What is more unusual about MacMOOSE compared to other MUD clients is its emphasis on supporting programming. Consider the program we discussed before: petting Rover. Here's how you would write the same program using the MacMOOSE client program. First, you'd click on the pencil icon (or select "Edit..." from the MOOSE menu) to indicate that you want to edit something.

Next, you'd enter the name of the object and the script you want to edit in the dialog box that appears (Figure 3.5).

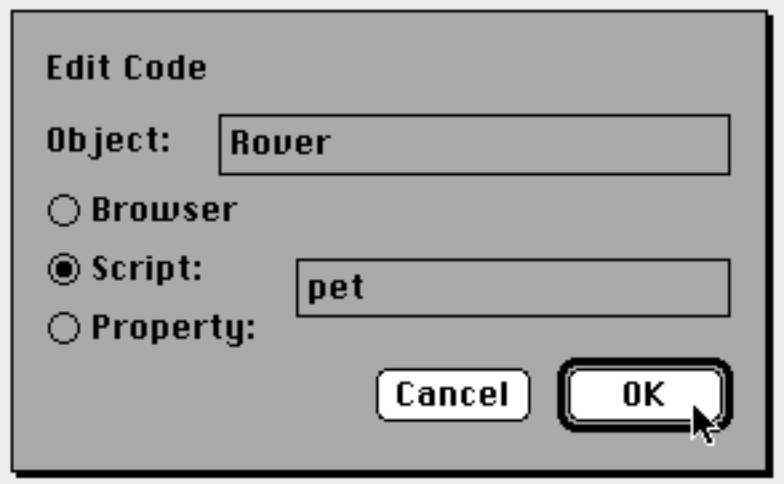

Figure 3.5: The MacMOOSE Edit Code Dialog Box

MacMOOSE asks if you want to add that script. Instead of having to remember a special command to declare a new script, the client does it for you. When you click OK, you get an editor, in which you can simply type your program (Figure 3.6).

In the script editor, you can change text in Macintosh WYSIWYG (What You See Is What You Get) style. If you wanted to change "his" to "her," you'd just click after "his," hit delete twice, and then type "er." Children are able to figure this out with little or no help. In version 1.0a1, feedback from compiling a script or verb appeared in the main window. This was awkward because you are not looking at the main window when you compile code. In version 1.0b1, we added a feedback area at the bottom of the editor windows, so your feedback is closely associated with your code. Screen real estate proved to be tight—you need adequate space to see the feedback but also adequate space left to edit code. The MOOSE compiler runs on a remote machine; the client runs on a local machine. We tried to minimize that gap by integrating compiler feedback with the client interface.

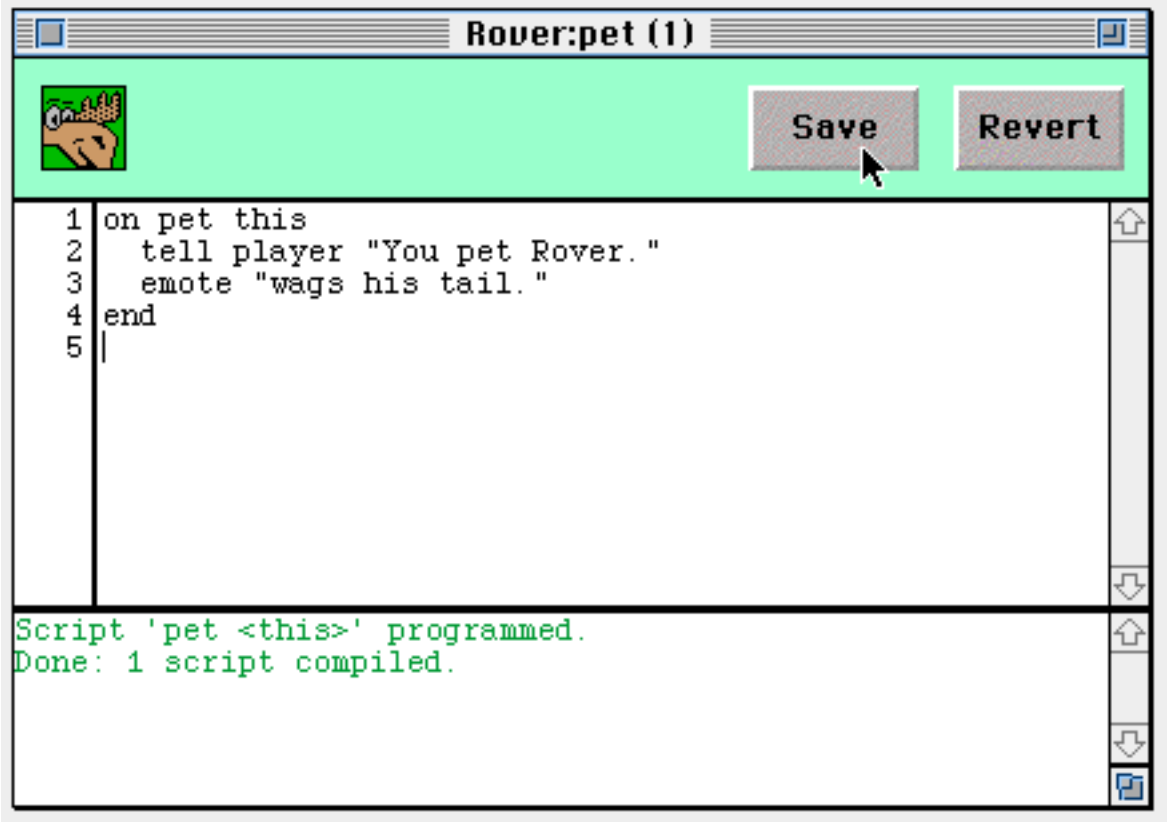

Figure 3.6: The MacMOOSE Script Editor

Unfortunately, we found that children often did not look at the feedback. Version 2.0 made positive feedback green and negative feedback red (Figure 3.7). Your most recent feedback is colored; feedback from previous compiles is turned black. While the children still may not always read the feedback, they know that red means something is wrong, and they notice the red text appearing. Additionally, the MOOSE icon[8] at the top of the window is turned red while the transaction is in progress and green again when it has completed. This gives users a good indication of when their compile is done. (Compiling a short script is immediate, but long ones can take several seconds.)

To edit an object, you can open up a browser on that object (Figure 3.8).

[8]The moose icon was designed by Michael Maier for MacMOOSE.

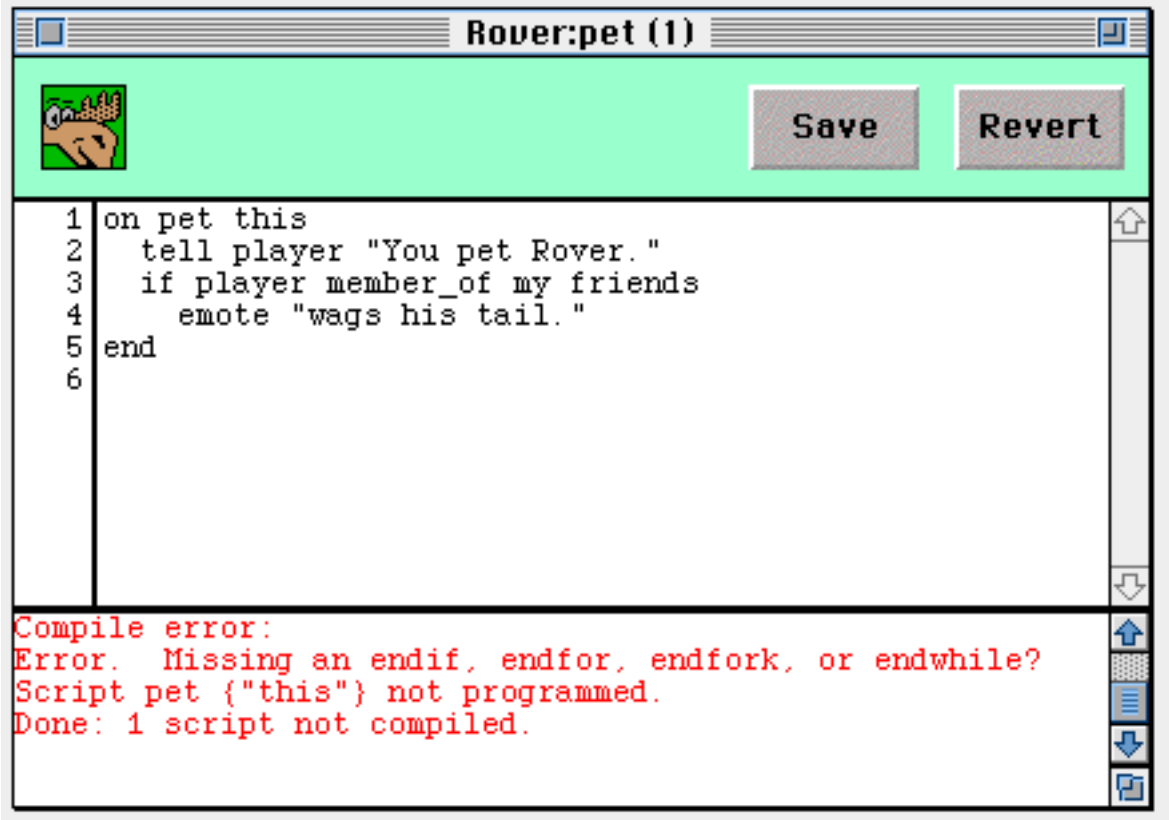

Figure 3.7: Feedback for a Compile Error

The central pop-up menu lets you climb the inheritance hierarchy to edit the object's parents. All of the object's scripts are shown on the left, and properties on the right. Double-clicking on a script or property opens an editor for that script or property. You may notice that fewer scripts are listed than properties. Script and property inheritance are handled slightly differently. If a parent object has a property, all of its children can have their own values for that property. If a parent object has a script, all of its child objects may use that script as it exists on the parent, but they can't have their own version of the script. For this reason, the only scripts listed are those declared on that object; properties listed include those declared on that object and on all of its parents. Adam Skwersky and I struggled with this interface design issue for over a month. The final solution significantly helps users work with these subtle differences in inheritance easily.

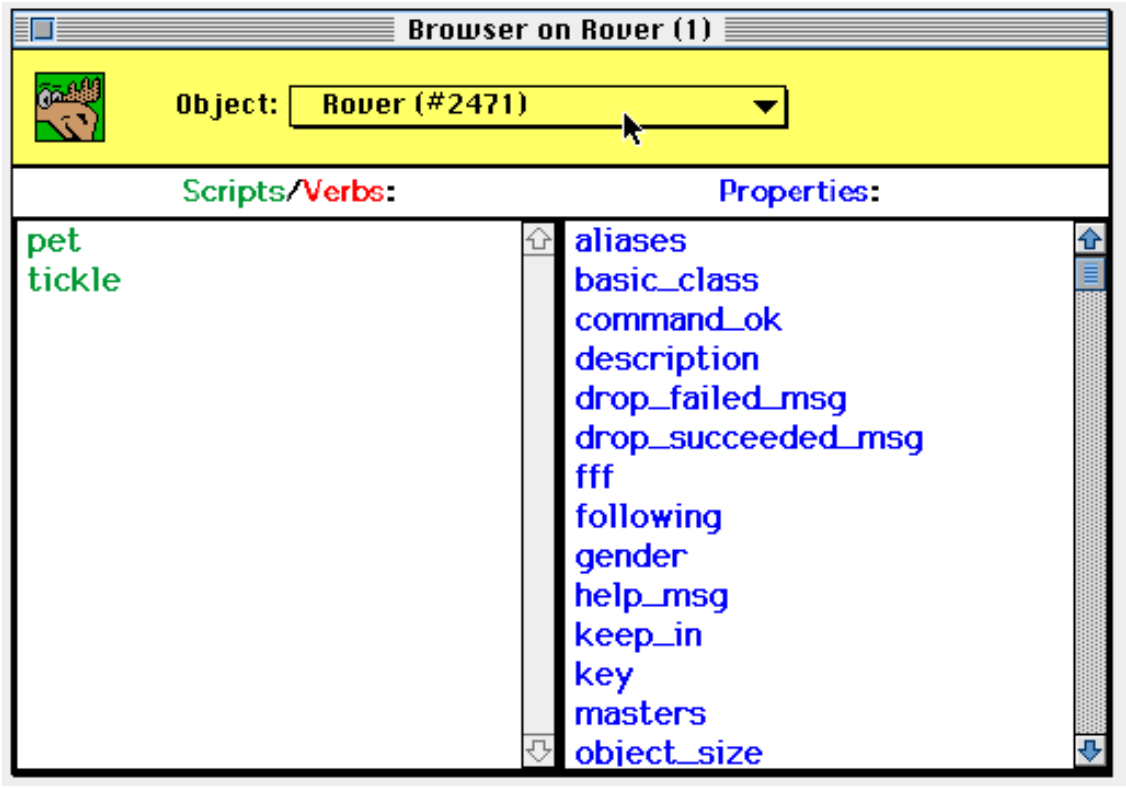

Figure 3.8: The MacMOOSE Object Browser

If you need help, you can get that help in a separate window (Figure 3.9). Multiple help messages can go in the same 'help browser'. This prevents help messages from scrolling away as you try to do what they recommend.

Finally, we added an interface to allow users to send MOOmail (mail internal to the virtual world) with WYSIWYG editing (Figure 3.10). The interface allows for sending mail, but not more advanced features like replying to a message including part of the previous message in the body.[9]

[9]The MacMOOSE documentation quips "no, we are not writing MOOdora."

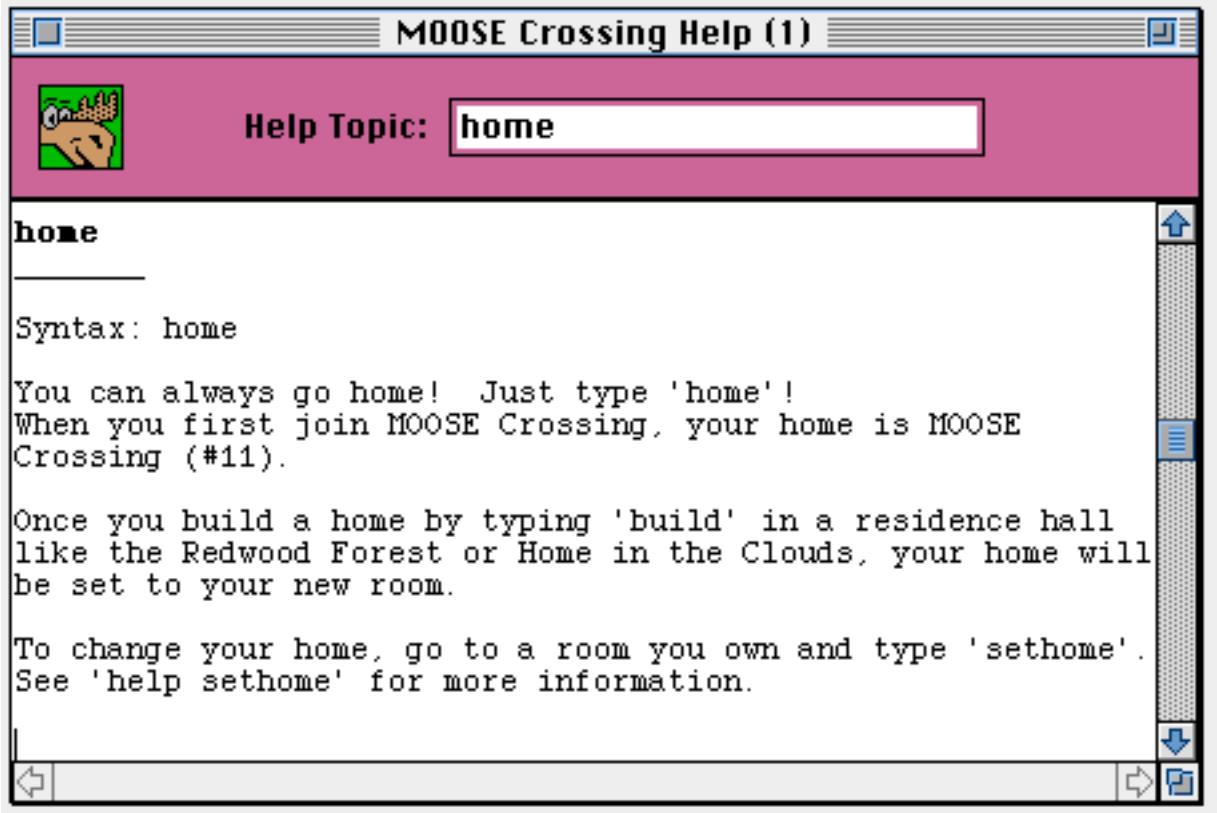

Figure 3.9: The MacMOOSE Help Browser

3.4.2 Equal Access for Few Versus Unequal Access for Many

MacMOOSE has proved to make learning to program significantly easier for children (and adults). While it is possible to access MOOSE Crossing without MacMOOSE, those doing so are at a significant disadvantage. Not wanting to create a class of haves and have-nots, I chose to restrict access to MOOSE Crossing to those who have access to Macintoshes on the Internet and can use MacMOOSE. This has unfortunately significantly limited the number of kids who have been able to use MOOSE Crossing. The number of children with access to Macintoshes is a small proportion of the total. Giving all participants equal access has made access available to a much more restricted group of children. That group is unfortunately disproportionately wealthy. We have been actively working with organizations such as CTCNet and PluggedIn who provide computing facilities to less advantaged children to try to broaden the demographic of children who have access to MOOSE Crossing. We hope to begin work on a Java version of MacMOOSE in the near future, which should make MOOSE Crossing accessible to a larger number of children.

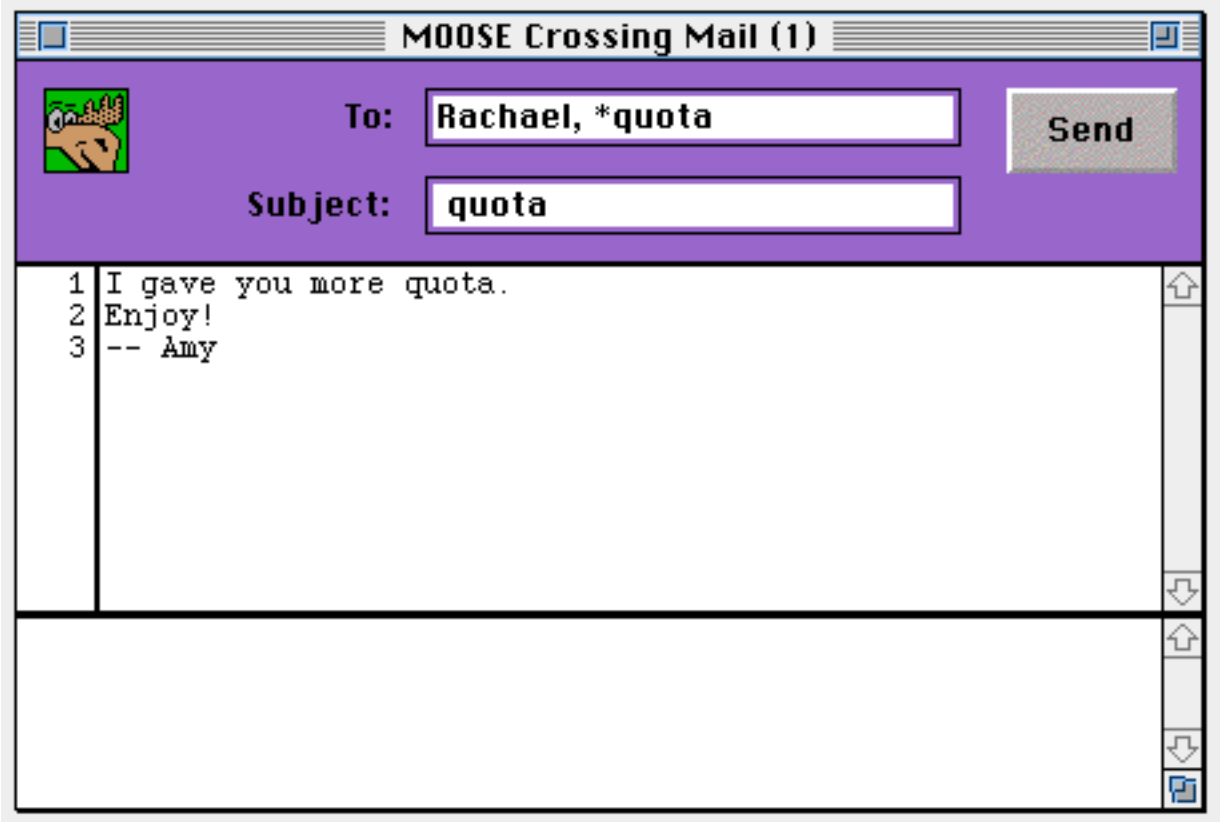

Figure 3.10: The MacMOOSE Mail Interface

3.5 Designing Empowering Technologies

Technology increasingly surrounds our everyday lives—the World Wide Web was only invented in the early 1990s, but by 1995 there were already URLs on bus ads. To what extent will the general public have meaningful control over those technologies? The answer is not clear. I believe that if you give people quality tools and social support for the use of those tools, they will surprise you with their intelligence and creativity. Part of taking users seriously involves including them in all stages of the design process: grounding design decisions in observations of real users rather than the formal concerns of professionals, and revising designs based on feedback from initial users. Users rise (or fall) to designers' expectations. We began the design of the MOOSE language and the MacMOOSE client with the assumption that kids are capable of great things. Our design agenda was also a political agenda—technology can and should empower people. I believe designing for nonprofessional users is a central issue for the future of computer science.